

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V
BRATISLAVE
STROJNÍCKA FAKULTA**

**MATICOVÉ OPERÁCIE NA PROGRAMOVATEĽNÝCH
MIKROOVLÁDAČOCH ARCHITEKTÚRY AVR**

Bakalárska práca

SjF-13432-87787

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V
BRATISLAVE
STROJNÍCKA FAKULTA**

**MATICOVÉ OPERÁCIE NA PROGRAMOVATEĽNÝCH
MIKROOVLÁDAČOCH ARCHITEKTÚRY AVR**

Bakalárska práca

SjF-13432-87787

Študijný program:	automatizácia strojov a procesov
Študijný odbor:	5.2.14. automatizácia
Školiace pracovisko:	Ústav automatizácie, merania a aplikovanej informatiky
Vedúci záverečnej práce:	doc. Ing. Gergely Takács, PhD.
Konzultant:	doc. Ing. Martin Gulán, PhD.

Bratislava, 2019

Mgr. Anna Vargová



ZADANIE BAKALÁRSKEJ PRÁCE

Študentka: **Mgr. Anna Vargová**
ID študenta: 87787
Študijný program: automatizácia a informatizácia strojov a procesov
Študijný odbor: 5.2.14. automatizácia
Vedúci práce: doc. Ing. Gergely Takács, PhD.
Konzultant: doc. Ing. Martin Gulán, PhD.
Miesto vypracovania: ÚAMAI Sjf, STU v Bratislave

Názov práce: **Maticové operácie na programovateľných mikroovládačoch architektúry AVR**

Jazyk, v ktorom sa práca vypracuje: slovenský jazyk

Špecifikácia zadania:

Úlohou je vyskúšať možnosti praktickej implementácie maticových operácií na osembitových programovateľných mikroovládačoch architektúry AVR. Maticové operácie sú často potrebné pre návrh a realizáciu riadiacich algoritmov, napríklad na riešenie Riccatiho rovnice pre výpočet optimálneho zosilnenia pre lineárne kvadratické (LQ) riadenie. Prvotnou úlohou predmetnej bakalárskej práce je preto preskúmať existujúce knižnice na maticové operácie a prešetriť nároky na pamäť a rýchlosť výpočtov, ak dané knižnice sú prekladané do exekutívy pre hardvér z rady mikroradičov AVR. Práca v druhom rade hľadá hranice nízkonákladového hardvéru na formuláciu univerzálnej knižnice pre LQ riadenie v prostredí Arduino IDE v jazyku C/C++, preto musí poskytovať prehľad algoritmov na riešenie Riccatiho rovnice, ich realizáciu v jazyku MATLAB a prehľad možností na ich prepis do C/C++ pre architektúru AVR.

Rozsah práce: cca. 30-50 s.

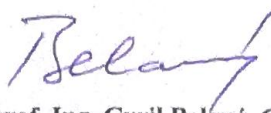
Riešenie zadania práce od: 11. 02. 2019

Dátum odovzdania práce: 24. 05. 2019

Mgr. Anna Vargová
študentka


prof. Ing. Cyril Belavý, CSc.
vedúci pracoviska

Slovenská technická univerzita
Ústav automatizácie, merania a aplikovanej informatiky
Strojnícka fakulta
Miestna časť
811 01 Bratislava
24. 05. 2019


prof. Ing. Cyril Belavý, CSc.
garant študijného programu

Čestné prehlásenie

Vyhlasujem, že som záverečnú prácu vypracovala samostatne s použitím uvedenej literatúry.

Bratislava, 24. mája 2019

.....
Vlastnoručný podpis

Chcela by som poďakovať vedúcemu diplomovej práce, doc. Ing. Gergelyovi Takácsovi, PhD, za jeho čas, trpezlivosť a inšpiratívne pripomienky a všetkým, ktorí mi pomáhali a stáli pri mne.

Bratislava, 24. mája 2019

Mgr. Anna Vargová

Názov práce: Maticové operácie na programovateľných mikroovládačoch architektúry AVR

Kľúčové slová: AVR, maticové operácie, LQR, Riccatiho rovnica, iteratívna metóda, Arduino

Abstrakt: Táto práca sa zaoberá možnosťami a obmedzeniami maticových operácií na programovateľných mikroovládačoch architektúry AVR. Skúma dostupné riešenia aplikácie maticových operácií na týchto mikropočítačoch, za použitia programátorského rozhrania Arduino IDE, kompilátorom avr-gcc. Hlbšie sa venuje niektorým dostupným knižniciam pre maticové operácie v prostredí C++, uvádza jednoduché testy knižníc zamerané na rýchlosť a spotrebu pamäte pri elementárnych maticových operáciách. Okrem maticových operácií všeobecne, obsahuje aj krátky prehľad stavovej reprezentácie systému a lineárne kvadratického riadenia, podrobnejšie sa venuje algoritmom na riešenie diskkrétnej algebraickej Riccatiho rovnice. V práci sú uvedené a reprodukované v MATLABe niektoré známe algoritmy riešenia diskkrétnej algebraickej Riccatiho rovnice. Iteratívny algoritmus je implementovaný na mikroovládač ATmega2560 z rodiny AVR čipov, z tohto algoritmu je následne vytvorená aj funkcia na výpočet LQ zosilnenia.

Title: Matrix operations on programmable microcontrollers with the AVR architecture

Keywords: AVR, matrix operations, LQR, Riccati equation, iterative method, Arduino

Abstract: This work deals with the possibilities and limitations of matrix operations on programmable microcontrollers with the AVR architecture, providing a research of available solutions of matrix operations implementation on these microcontrollers using Arduino IDE programming interface with avr-gcc compiler. The paper also contains a deeper analysis of some available C++ libraries designed for work with matrices, including several simple performance tests, especially focused on program speed and memory usage during elementary matrix operations. Beside matrix operations in general introduces a brief overview of state-space representation and linear quadratic regulator, describes the discrete algebraic Riccati equation more in detail. Some algorithms for solving discrete Riccati equation are mentioned and reproduced in MATLAB. An iterative solution algorithm is implemented on microcontroller ATmega2560 from AVR family. From this algorithm a function to calculate the LQ gain on AVR is created.

Obsah

Úvod	1
1 Architektúra AVR	2
1.1 Pamäť architektúry AVR	2
1.2 Práca s desatinnými číslami na AVR	3
2 Knižnice pre maticové operácie	6
2.1 Knižnica MatrixMath	7
2.2 Knižnica Matrix	8
2.3 Knižnica BasicLinearAlgebra	9
2.4 Knižnica Eigen	9
2.5 Porovnanie knižníc	9
3 Lineárne kvadratická úloha	14
3.1 Stavový model	14
3.2 Lineárne kvadratický regulátor	15
4 Riccatiho rovnica	17
4.1 Algoritmy na riešenie diskkrétnej algebraickej Riccatiho rovnice	17
4.1.1 Algoritmus „Jack Benny“	17
4.1.2 Algoritmus zovšeobecnených vlastných čísel	18
4.1.3 Schurova metóda	20
4.1.4 Nerekurzívna metóda	21
4.2 Porovnanie algoritmov na riešenie Riccatiho rovnice	21
5 Algoritmus „Jack Benny“ na mikrovládačoch s architektúrou AVR	25
5.1 Funkcia na výpočet LQ zosilnenia	27
5.2 Príklad použitia funkcie	29
6 Záver	33
Literatúra	35

Zoznam obrázkov

2.1	Porovnanie využitia pamäte pri výpočtoch jednotlivými knižnicami .	11
2.2	Porovnanie potrebného času pri výpočtoch jednotlivými knižnicami .	12
4.1	Konvergencia prvkov matice P	23
5.1	Rozdiel medzi MATLABom a BLA	27
5.2	Rozdiel medzi MATLABom a BLA predelený očakávanou hodnotou .	27
5.3	Príklad satelitu	30
5.4	Simulácia v MATLABe a v BLA	32

Zoznam tabuliek

1.1	Porovnanie Heap a Stack pamäte [12]	3
1.2	Porovnanie jednoduchých operácií s rôznymi dátovými typmi na AVR [6]	4
1.3	Porovnanie zložitejších operácií s rôznymi dátovými typmi na AVR [6]	4
1.4	Konverzia dátových typov na AVR [6]	4
2.1	Knižnica MatrixMath	8
2.2	Knižnica Matrix	9
2.3	Knižnica BasicLinearAlgebra	10
2.4	Knižnica Eigen	10
4.1	Výsledky v MATLABe	22
4.2	Rozdiely medzi prvkami matíc P^{ref} a P^{func}	22
4.3	Porovnanie algoritmov v MATLABe pomocou príkazu profiler	24
5.1	Výsledky v MATLABe	26

Úvod

Cieľom tejto práce je preskúmanie dostupných možností maticových operácií na mikroovládačoch architektúry AVR. Problémy, ktorých riešenie si vyžaduje použitie maticových operácií, sa často vynárajú v teórii riadenia, ale aj v iných odboroch, čo vedie k týmto výpočtom aj na programovateľných mikroovládačoch. Architektúra AVR je jednoduchým a bežným typom mikroovládačov v praxi, pri výučbe predmetov týkajúcich sa automatizácie a programovania, ale aj vo voľnom čase, možno aj vďaka existujúcemu projektu Arduino, ktorý zahŕňa celý rad hardvérových a softvérových riešení programovateľných mikroovládačov. Veľa prototypizačných dosiek z tohto projektu využíva práve MCU s architektúrou AVR. Pri tvorbe tejto práce je teda hlavne použitá prototypizačná doska Arduino Mega 2560 s mikroovládačom ATmega2560. Na programovanie mikroovládača je použité programátorské rozhranie Arduino IDE, s kompilátorom `avr-gcc`. Práca s maticami v dialektoch jazyka C/C++ (ktorý sa používa aj v Arduino IDE) dokáže byť bez použitia správnej knižnice sama o sebe náročná. Navyše niektoré špecifické vlastnosti mikroradičov AVR môžu byť tiež obmedzujúce, hlavne štruktúra pamäte AVR a práca s dátovým typom `float` bez hardvérovej podpory použitia tohto dátového typu. Kvôli týmto skutočnostiam je uvedená aj analýza dostupných softvérových riešení — knižníc pre prácu s maticami v C++, ktoré sa dajú jednoducho použiť aj v programátorskom prostredí Arduino IDE — a jednoduché testovanie ich výkonnosti. Napriek tomu, že by sa táto tematika mohla zdať dostatočne pokrytá v literatúre, v skutočnosti sa javí ako pomerne málo zdokumentovaná oblasť. Snahou tejto práce je preto poskytnúť stručný prehľad problematiky. Práca sa teda nevenuje len elementárnym maticovým operáciám, ale aj výkonu zložitejších algoritmov zahrňujúcich maticové operácie na mikroovládačoch AVR. Dobrým príkladom zložitejších algoritmov môže byť riešenie diskkrétnej algebraickej Riccatiho rovnice, ako častého matematického problému, uvedené aj v tejto práci. Niekoľko algoritmov riešenia diskkrétnej algebraickej Riccatiho rovnice sú popísané a zreplikované v MATLABe a jedna iteratívna metóda je aj implementovaná na mikroovládač s architektúrou AVR s uvedeným príkladom použita a s tvorbou funkcie z tohto algoritmu.

1 Architektúra AVR

V tejto kapitole si stručne uvedieme niektoré vlastnosti špecifické pre architektúru AVR, ktoré môžu ovplyvniť implementáciu maticových operácií na mikroovládače s takouto architektúrou. Sú to hlavne vlastnosti týkajúce sa pamäte mikroovládača, nakoľko plné matice sú štruktúry s pomerne veľkou spotrebou pamäte, ktorej mikroovládače nemávajú veľa nazvyš. Druhou obmedzujúcou skutočnosťou pri výkone výpočtov je neexistujúca hardvérová podpora operácií s desatinnými číslami (Floating Point Unit, FPU). Kvôli dostupnosti a rozvinutej podpore — vo forme ľahko použiteľného programátorského rozhrania, softvérových a hardvérových doplnkov — v ďalších úvahách sa obmedzíme na porotypizačnú dosku Arduino Uno, resp. Arduino Mega, obsahujúcu MCU s architektúrou AVR, s použitím programátorského rozhrania Arduino IDE, s kompilátorom `avr-gcc`.

1.1 Pamäť architektúry AVR

Mikroovládače z rodiny AVR používajú upravenú Harvardovskú architektúru pamäte, pozostávajúcu z troch typov pamäte:

- programová pamäť, alebo tzv. Flash pamäť,
- statická pamäť (SRAM) a
- elektricky mazateľná pamäť (EEPROM).

Programová pamäť sa používa predovšetkým na ukladanie obrazu programu a všetkých inicializovaných údajov. Kód uložený v programovej pamäti sa dá kedykoľvek spustiť, ale dáta uložené v tejto časti sa pomocou spusteného programu modifikovať nedajú. Ak by sme chceli tieto údaje upraviť, musíme si ich najskôr skopírovať do statickej pamäte.

Statická pamäť je časť pamäte, z ktorej a do ktorej je možné zapisovať údaje z bežiacего programu. Pri behu programu sa používa viacerými spôsobmi. Na uloženie statických dát — je to blok pamäte vyhradený na ukladanie všetkých globálnych a statických premenných. Ako tzv. halda, resp. heap — slúži na ukladanie dynamicky pridelených dátových položiek. Ako zásobník (stack) — určený pre lokálne premenné, záznamy o prerušení a volaní funkcií. Problémy s pamäťou vznikajú často kvôli kolíziám heap a stack pamäte [8], preto je dôležité kedy a ako ich použiť pri tvorbe programu, pričom aj jedno, aj druhé majú svoje pre a proti. V Tab. 1.1, ktorú uvádza aj stránka Geeks for Geeks [12], vidíme stručné porovnanie vlastností týchto dvoch typov pamäte.

PARAMETER	HEAP	STACK
Postup pridelenia pamäťového priestoru	Náhodne	Priebežne, v ucelených blokoch
Pridelenie a uvoľňovanie pamäťového priestoru	Manuálne, programátorom	Automaticky, podľa inštrukcií kompilátora
Implementácia	Jednoduchá	Ťažká
Prístup k dátam	Pomalšie	Rýchlejšie
Hlavný problém	Fragmentácia pamäte	Nedostatok pamäte
Princíp lokálnosti	Primeraný	Vynikajúci
Flexibilita	Zmena veľkosti možná	Pevná veľkosť

Tabuľka 1.1: Porovnanie Heap a Stack pamäte [12]

Elektricky mazateľná pamäť je ďalšia forma permanentnej pamäte, ktorá môže byť čítaná a prepisovaná bežiacim programom, dáta z nej sa však dajú čítať len po bajtoch, jej použitie je teda trochu obtiažnejšie a pomalšie ako pamäte SRAM.

1.2 Práca s desatinnými číslami na AVR

Ako sme už spomínali, AVR architektúra neobsahuje hardvérovú podporu operácií s desatinnými číslami. Desatinné čísla sú zastúpené ako čísla s plávajúcou desatinnou čiarkou (floating point number), podľa normy IEEE 754 [14]. Táto norma sa riadi tromi jednoduchými pravidlami:

- znamienko mantisy — 0 predstavuje kladné číslo, zatiaľ čo 1 záporné,
- znamienkový exponent — pole exponentov musí obsiahnuť kladné aj záporné exponenty,
- normalizovaná mantisa — mantisa je časťou čísla vo vedeckom zápise, ktorá sa skladá z jeho platných číslic a to tak, že len jedno nenulové číslo sa nachádza od desatinnej čiarky vľavo.

Dôsledkom tohto je zložitejší algoritmus, dokonca aj pri jednoduchých operáciách ako sú sčítavanie alebo násobenie. Snahou je teda pri výpočtoch vyhnúť sa použitiu desatinných čísel všade, kde to je možné. Na druhej strane vďaka väčšiemu rozlíšeniu, slúžia na lepšie zobrazenie spojitých javov. Dátový typ používaný na ukladanie desatinných čísel na prototypizačných doskách Arduino Uno/Mega je 4 bajtový `float`, s presnosťou na 6-7 číslic, presnejší typ `double` táto platforma nerozoznáva, a pracuje s ním ako s `floatom`. Do akej miery je práca s `floatmi` náročnejšia ako práca s ostatnými dátovými typmi sa dá ukázať pomerne jednoducho. Podrobný popis a výsledky môžeme nájsť napríklad aj na stránke organizácie Arduino [6]. Vybrané výsledky meraní vrátane ukladania premenných na RAM sú

uvedené v Tab. 1.2, 1.3 a 1.4. Zaujímavé sú hlavne výsledky z Tab. 1.3, z ktorej vidíme, že AVR na zložitejšie operácie využíva interpolačné tabuľky, čoho dôsledkom je rádovo podobná dĺžka operácií bezohľadu na použitý dátový typ. Dokonca pre tieto operácie je dátový typ `float` výhodnejší, nakoľko v tomto prípade nie je nutné pretypovanie premennej počas a po výpočtoch.

	Byte		Int		Long		Float	
	μs	Cyklov CPU	μs	Cyklov CPU	μs	Cyklov CPU	μs	Cyklov CPU
Zápis na RAM	0.13	2	0.25	4	0.50	8	0.50	8
$z=x+y$	0.44	7	0.88	14	1.76	28	8.11	130
$z=x-y$	0.44	7	0.88	14	1.76	28	8.15	130
$z=x*y$	0.63	10	1.50	24	4.53	72	9.67	155
$z=x/y$	12.9	207	15.7	251	40.9	654	31.1	497
$z=x\%y$	12.9	207	15.7	251	40.9	654	31.1	497
$z=\min(x,y)$	0.73	12	1.29	21	2.62	42	5.33	85
$z=\max(x,y)$	0.73	12	1.29	21	2.62	42	5.33	85
$z=\text{abs}(x), x>0$	-	-	1.07	17	2.14	34	5.08	81
$z=\text{abs}(x), x<0$	-	-	1.24	20	2.50	40	5.08	81

Tabuľka 1.2: Porovnanie jednoduchých operácií s rôznymi dátovými typmi na AVR [6]

	Byte		Int		Long		Float	
	μs	Cyklov CPU	μs	Cyklov CPU	μs	Cyklov CPU	μs	Cyklov CPU
$z=\text{sqrt}(x)$	52.2	835	51.2	819	52.7	843	45.8	732
$z=\sin(x)$	121.3	1941	126.4	2023	146.2	2339	123.8	1981
$z=\cos(x)$	120.7	1931	130.6	2089	146.2	2339	123.1	1969
$z=\tan(x)$	148.1	2369	157.1	2514	176.0	2816	148.3	2373
$z=\text{pow}(x,y)$	309.0	4944	306.8	4908	342.9	5487	148.3	4828

Tabuľka 1.3: Porovnanie zložitejších operácií s rôznymi dátovými typmi na AVR [6]

	Byte		Int		Long		Float	
	μs	Cyklov CPU	μs	Cyklov CPU	μs	Cyklov CPU	μs	Cyklov CPU
$z=\text{int}(x)$	0.44	7	-	-	0.752	12	4.85	78
$z=\text{long}(x)$	0.82	13	1.00	16	-	-	5.17	83
$z=\text{float}(x)$	3.22	52	3.89	62	4.46	71	-	-

Tabuľka 1.4: Konverzia dátových typov na AVR [6]

Tabuľky 1.2, 1.3 a 1.4 dobre znázorňujú dôležitosť efektívnych algoritmov, pri jednotlivých operáciách, hlavne pri dátovom type `float`, kde pri úkonoch musí algoritmus vyčítať mantisu aj exponent, pri operáciách s viacerými `floatmi` dokonca ich priviesť „na spoločného menovateľa“. Ak ale MCU musí tieto činnosti vykonávať na celom poli `floatov` — akým môžeme chápať matice, ktorých prvkami sú desatinné čísla — je celkom prirodzené, že ich počet sa niekoľkokrát znásobuje a narastá tým aj čas potrebný na výpočet, aj spotrebovaná pamäť. Význam vhodne zvolených operácií a algoritmov je teda samozrejímavý, práve preto sa v Kap. 2 budeme venovať výberu vhodnej knižnice pre maticové operácie na mikroovládačoch architektúry AVR.

2 Knížnice pre maticové operácie

V tejto kapitole sa budeme venovať predovšetkým dostupným možnostiam pri riešení maticových operácií na AVR, špeciálne pre použitie v prostredí Arduino IDE pri kompilácii na prototypizačnú dosku Arduino Mega 2560 alebo Arduino Uno.

Jazyky C/C++ samy o sebe neposkytujú dostatočnú podporu maticových operácií, hoci práca s maticami môže byť veľmi užitočná, až nevyhnutná pri riešení niektorých úloh z teórie riadenia. Je teda veľmi dôležité mať správnu knižnicu pre maticové operácie. Navyše okrem chýbajúcej podpory v jazykoch C/C++ nesmieme zabudnúť na možné komplikácie spojené s architektúrou AVR.

Pri otázke „Akú knižnicu použiť?“ sa naskytáva hneď niekoľko možných odpovedí. Môžeme si napríklad knižnicu sami naprogramovať. Písanie vlastných knižníc je ale časovo náročné a navyše často aj neefektívne [15]. Oproti tomu výhodou knižnice od „tretej strany“ okrem rýchlosti implementácie je tiež lepšie otestovaný kód. Ani výber hotovej knižnice však nie je jednoznačný, nakoľko existuje množstvo riešení určených pre jazyky C alebo C++, ako sú napríklad knižnice Armadillo [1], BLAS [2], Blaze [3], LAPACK [4], LELA [13]. Treba ale brať na vedomie, že pri programovaní mikropočítača nemusíme mať k dispozícii všetky vlastnosti týchto jazykov. Ak by sme chceli tieto knižnice použiť napríklad pri programovaní Arduina pomocou prostredia Arduino IDE, museli by sme spraviť niekoľko nevyhnutných úprav knižníc, resp. doprogramovať vhodné rozhranie.

Našťastie existuje aj množstvo knižníc, ktoré sú už upravené pre použitie na doskách Arduino, preto sa budeme v tejto práci venovať tejto alternatíve. Uvedieme štyri — prakticky najdostupnejšie — takéto knižnice a výsledky z ich testovania:

- MatrixMath,
- Matrix,
- BasicLinearAlgebra (BLA) a
- Eigen.

Existuje mnoho postupov na testovanie výkonnosti programov, návodov na správny benchmarking, pre účely tejto práce je ale úplne postačujúce vykonanie niekoľko výpočtov s použitím základných maticových operácií. Nižšie popísané testovanie je zamerané len na rýchlosť výpočtov a zaberanie pamäte pri rôznych operáciách, nakoľko možnosti mikroovládačov sú v tomto smere obmedzené a pre použitie v praxi je veľmi dôležité narábať efektívne s týmito zdrojmi.

Postup pri testovaní:

- Výber testovaných úkonov:
 - uloženie náhodnej matice,
 - uloženie náhodnej matice a jednotkovej matice,
 - sčítanie dvoch náhodných matíc,
 - odčítanie,
 - násobenie matíc,
 - násobenie matice vektorom zľava,
 - násobenie matice vektorom sprava,
 - transpozícia matice a
 - inverzia.
- Pomocou MATLABu sme vygenerovali náhodné matice a vektor, ktorých elementami sú desatinné čísla (float), uvedené nižšie, matica I označuje jednotkovú diagonálnu maticu príslušných rozmerov.

Náhodné matice A, B a náhodný vektor b

$$A = \begin{bmatrix} 0.81 & 0.91 & 0.28 \\ 0.91 & 0.63 & 0.55 \\ 0.13 & 0.10 & 0.96 \end{bmatrix}, b = \begin{bmatrix} 0.96 & 0.96 & 0.14 \\ 0.16 & 0.48 & 0.42 \\ 0.97 & 0.80 & 0.92 \end{bmatrix} B = \begin{bmatrix} 0.96 \\ 0.96 \\ 0.14 \end{bmatrix}$$

- Odmeranie času a pamäte potrebného na zbehnutie „doplnkového“ kódu — `void()`, `loop()`, `for()` cyklus a vypísanie odmeraného času na sériový port. Čas bol meraný pomocou funkcie `micros()`, použitú pamäť hlási Arduino IDE pri kompilácii.
- Pri testovaní každej knižnice sa premenné deklarovali ako globálne, nestatické, aby sa zamedzilo optimalizácií kódu kompilátorom.
- Vykonanie každého testovaného úkonu stokrát (v rámci `for()` cyklu) a po odčítaní hodnoty „doplnkového“ kódu a predelení počtom opakovaní, získanie priemerného času a využitej pamäte.

2.1 Knižnica MatrixMath

Knižnicu MatrixMath môžeme nájsť aj medzi ponúkanými knižnicami Arduino IDE. Je to minimálna knižnica pre lineárnu algebru na Arduino. Matice sú zastúpené ako jednoduché dvojrozmerné polia, ktorého elementami sú dáta typu `double` (resp. `float`). Zhoda rozmerov matíc sa musí kontrolovať manuálne. Syntax knižnice je trochu ťažkopádna v dôsledku jednoduchej štruktúry triedy `Matrix`. Má len minimálne množstvo metód, pričom zaujímavými funkcionalitami sú násobenie skalárom

	Pamät (B)	Globálne premenné (B)	Čas na 100 meraní (ms)	Priemerný čas (ms/100 meraní)
Uloženie matice A	2082	188	72*	72
Uloženie matíc A a I	2082	188	72*	72
$A + B$	2966	296	9900	99
$A - B$	2966	296	10372	103.72
$A \cdot I$	3202	296	41432	414.32
$A \cdot B$	3254	296	50772	507.72
$A \cdot b$	3070	248	17560	175.6
$b^T \cdot A$	3074	248	17976	179.76
A^T	2380	260	2472	24.72
A^{-1}	3974	264	67796	677.96

*optimalizácia kompilátora

Tabuľka 2.1: Knížnica MatrixMath

a kopírovanie matíc — ktoré v tejto štruktúre je v skutočnosti nutnosťou, ktorá vyplýva z vlastností C/C++ pri práci s poliami. Ako užitočná metóda (ktorá v tomto prostredí napriek očakávaniam nie je samozrejmosťou) sa ukázalo odoslanie matice na sériový port. Tu by sme možno navrhli tvorcom tejto knižnice úpravu zobrazenia matice, nakoľko sériový monitor má obmedzenú veľkosť zobrazených čísel, pri manipulácii s maticami, ktorých prvkami sú floaty môže dôjsť k pretečeniu. Vhodným vylepšením by možno bolo pretypovanie zobrazených prvkov, alebo „simulovanie“ vedeckého zobrazenia čísel.

Počas testovania knižnice sa pri ukladaní tej istej náhodnej matice uvedenou metodikou testovania sa nám nepodarilo úplne obísť optimalizáciu kompilátora: pri stonásobnom ukladaní bol čas potrebný na tento úkon rovnaký, ako čas pri jednom uložení matice. Toto na jednej strane znehodnocuje meranie, na druhej strane vnímame pozitívne, že pri použití tejto knižnice sa kompilátor snažil vyhnúť zbytočnej práci. Pri testovaní ostatných knižníc sa nič podobné nevyskytlo. Výsledky meraní, sú uvedené v Tab. 2.1.

2.2 Knížnica Matrix

Táto knížnica je tiež jednoduchou knižnicou ponúkajúcou malé množstvo funkcií, ale s oveľa prijateľnejšou a intuitívnejšou syntaxou, vďaka trošku zložitejšej štruktúre založenej na šablónach — template library. Neplatné operácie (napríklad kvôli nesúladu rozmerov matíc) v prípade knižnice Matrix vedú k prázdnej matici. Umožňuje používať operátory $+$, $-$, $/$, $*$ tak, ako sme zvyknutí v iných programovacích jazykoch. Metódy transpozície a inverzie sa volajú ako `Matrix<Type>::transpose()` a `Matrix<Type>::inv()`, kde `Type` označuje inicializovanú maticu [7]. Výsledky meraní, sú uvedené v Tab. 2.2.

	Pamät (B)	Globálne premenné (B)	Čas na 100 meraní (ms)	Priemerný čas (ms/100 meraní)
Uloženie matice A	2886	234	7172	71.72
Uloženie matíc A a I	3122	234	16268	162.68
$A + B$	3996	302	21068	210.68
$A - B$	3994	300	21588	215.88
$A \cdot I$	4184	276	59412	594.12
$A \cdot B$	4358	312	82856	828.56
$A \cdot b$	4284	288	30092	300.92
$b^T \cdot A$	4284	288	30092	300.92
A^T	3282	234	11476	114.76
A^{-1}	5608	264	220880	2208.8

Tabuľka 2.2: Knižnica Matrix

2.3 Knižnica BasicLinearAlgebra

Netreba si ju mýliť s klasickou BLAS knižnicou. BasicLinearAlgebra je už trošku sofistikovanejšou knižnicou, ktorá definuje triedu dvojrozmerných matíc (Matrix) s ľubovoľným počtom riadkov, stĺpcov a tiež typu, dokonca spôsobu ukladania do pamäte. Tak ako aj knižnica Matrix, aj BasicLinearAlgebra umožňuje použitie bežných operátorov $+$, $-$, $/$, $*$ prirodzene. Pre transpozíciu a inverziu matice ponúka táto knižnica aj viacero možností. Okrem toho umožňuje aj ďalšie pokročilejšie operácie matíc. Okrem toho, táto knižnica obsahuje aj kontrolu správnosti rozmerov a v prípade nekonzistentnosti vyhlási chybu pri kompilácii [18]. Tak, ako knižnica Matrix, je aj BLA šablónovou knižnicou. Výsledky meraní sú uvedené v Tab. 2.3.

2.4 Knižnica Eigen

Knižnica Eigen je široko akceptovaná knižnica C++ pre lineárnu algebru: matice, vektory, numerické riešenia a algoritmy s nimi súvisiace. Na GitHubu [16] je dostupná modifikácia knižnice Eigen 3.0.6 s úpravami a rozšíreniami, ktoré umožňujú používať túto knižnicu na doske Arduino. Okrem výhod ponúkaných knižnicou BasicLinearAlgebra, táto knižnica ponúka ďalšie funkcionality, ktoré by sa nám pri tvorbe LQR knižnice mohli hodiť. Nevýhodou tejto knižnice je, že okrem nej potrebujeme do kódu zahrnúť aj hlavičkový súbor `stdport.h`, ktorý nahrádza v jazyku C++ dobre známu knižnicu `iostream.h`. Výsledky meraní, sú uvedené v Tab. 2.4.

2.5 Porovnanie knižníc

Po prvotnom testovaní knižníc môžeme porovnať získané výsledky. Na Obr. 2.1 a 2.2 vidíme, že knižnica Matrix spotrebuje pri výpočtoch najviac pamäte a tiež sa

	Pamät (B)	Globálne premenné (B)	Čas na 100 meraní (ms)	Priemerný čas (ms/100 meraní)
Samotná knižnica	656	9		
Uloženie matice A	2598	188	18288	182.88
Uloženie matíc A a I	2598	188	18288	182.88
$A + B$	2958	188	36876	368.76
$A - B$	2958	188	36876	368.76
$A \cdot I$	2902	188	36696	366.96
$A \cdot B$	2958	188	36876	368.76
$A \cdot b$	2598	188	18288	182.88
$b^T \cdot A$	2802	188	24448	244.48
A^T	2598	188	18288	182.88
A^{-1}	4110	192	95352	953.52

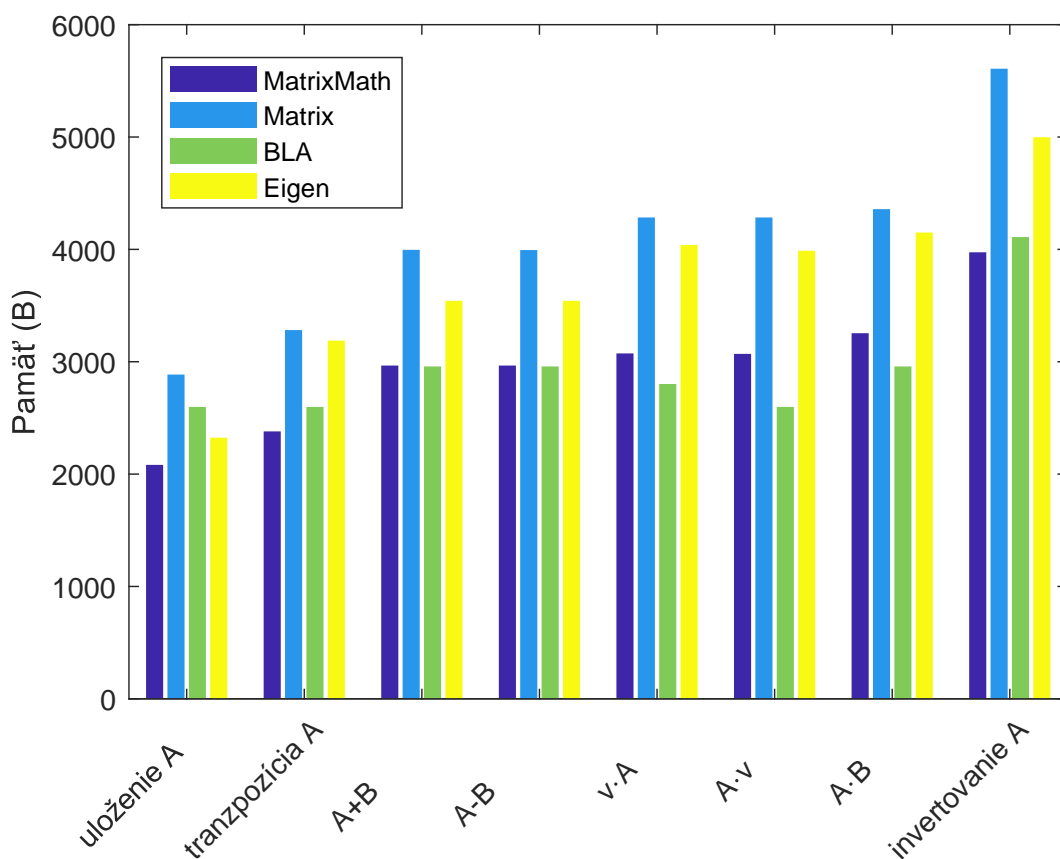
Tabuľka 2.3: Knižnica BasicLinearAlgebra

	Pamät (B)	Globálne premenné (B)	Čas na 100 meraní (ms)	Priemerný čas (ms/100 meraní)
Uloženie matice A	2324	188	3840	38.4
Uloženie matíc A a I	2504	188	7740	77.4
$A + B$	3542	198	8160	81.6
$A - B$	3542	198	8592	85.92
$A \cdot I$	4142	198	38984	389.84
$A \cdot B$	4150	198	48180	481.8
$A \cdot b$	3988	198	15560	155.6
$b^T A$	4040	198	16428	164.28
A^T	3188	198	1856	18.56
A^{-1}	4998	198	41588	415.88

Tabuľka 2.4: Knižnica Eigen

ukázala ako najpomalšia pri invertovaní matice. Pri skúmaní tejto knižnice, resp. jej využitia v budúcnosti treba poznamenať, že je stále vo vývoji, do ďalších úvah by sme ju nezahrnuli.

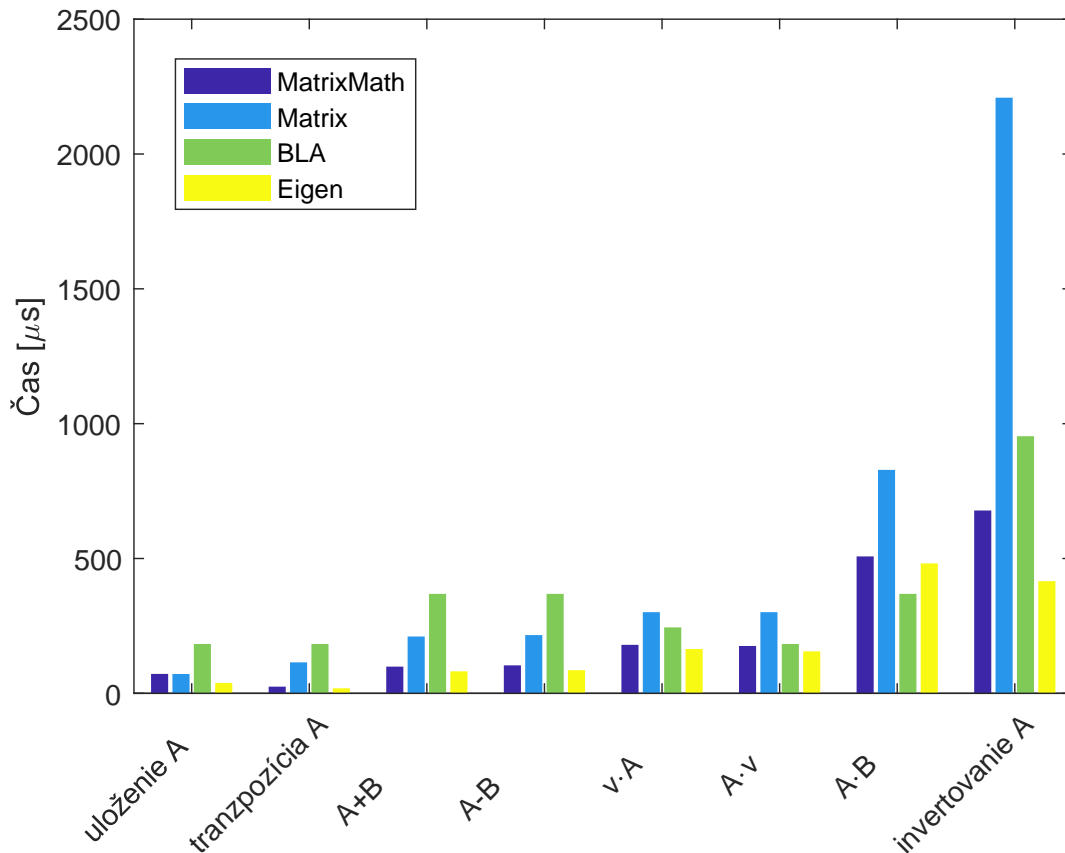
Pri porovnaní spotreby pamäte ostatných knižníc zistíme, že pri väčšine skúmaných operácií knižnica Eigen spotrebuje najviac pamäte, na druhej strane tieto výpočty vykonáva v najkratšom čase, kým knižnice BLA a MatrixMath spotrebávajú pri výpočtoch najmenej pamäte, a navyše knižnica MatrixMath sa javí porovnateľne rýchla, ako knižnica Eigen (okrem operácie invertovania matice). Avšak rozdiely medzi uvedenými knižnicami nie sú také veľké, aby sme mohli jednoznačne určiť najvhodnejšiu pre prácu s maticami na Arduino z nich, resp. by sa mohlo zdať, že pre jednoduché operácie sú vhodné všetky. Je tu však ešte jeden problém, týkajúci sa knižnice Eigen.



Obr. 2.1: Porovnanie využitia pamäte pri výpočtoch jednotlivými knižnicami

Napriek tomu, že sa zdá knižnica Eigen široko využiteľná (a často aj odporúčaná na fórach), hlavne vďaka ľahkému používaniu a širokej škále zahrnutých metód, ako je výpočet vlastných čísel, vlastných vektorov, dokonca Schurova dekompozícia alebo aj Kalmanov filter, použitie tejto knižnice na architektúre AVR naráža na niekoľko problémov. Aby sme si objasnili tieto problémy, je potrebné sa pozrieť na knižnicu bližšie. Knižnica Eigen má vopred definované veľkosti statických matic¹, od jedna po

¹Statická matica v tomto prípade označuje maticu, ktorej veľkosť počas jednotlivých výpočtov



Obr. 2.2: Porovnanie potrebného času pri výpočtoch jednotlivými knižnicami

štyri, či do počtu riadkov alebo stĺpcov. Na zápis matice, ktorej počet riadkov alebo stĺpcov je väčší ako štyri je potrebné túto maticu definovať ako dynamickú maticu², aj napriek tomu, že sa jej veľkosť počas výpočtov meniť nemusí [16]. Hlavný rozdiel medzi maticami inicializovanými ako statické alebo dynamické spočíva v ukladaní veľkosti matice do pamäti. Základná myšlienka je taká, že veľkosť statickej matice by sa mala zapísať do stack pamäte a veľkosť dynamickej matice by sa mala zapísať do heap pamäte. Napriek tomu, že aj stack aj heap sú súčasťou SRAMu, táto logika môže sťažiť, dokonca znemožniť niektoré výpočty s dynamickými maticami [9], napríklad násobenie dynamických matíc sa na dosku Arduino Uno/Mega skompiluje bez chybového hlásenia, ale výpočet sa nevykoná. Navyše toto rozdielne uloženie je zabezpečené použitím pamäťovej triedy `mutable storage class`, známej iba v dialekte C++, ktorú kompilátor Arduino IDE pri kompilácii na dosku Arduino Uno/Mega vôbec nemusí pochopiť. Tu by sme poznamenali, že pri kompilácii krátkeho testovacieho programu (minimal working example) na dosku Arduino Due, ktorej procesor má inú architektúru (ARM) sa tieto problémy neobjavili.

Po týchto úvahách sa dostávame k dvom knižniciam vhodným na použitie na

nemení.

²Dynamická matica v tomto prípade označuje maticu, ktorej veľkosť sa počas výpočtov môže meniť

procesoroch AVR: MatrixMath a BLA. Skúsme si však na chvíľu predstaviť, čo by sa stalo, keby sme okrem jednoduchých maticových operácií s maticami, ktorých prvky sú typu `float`, chceli niečo viac, zložitejšie výpočty, napríklad na riadenie procesu, alebo dokonca vytvárať vlastné knižnice riadenia. V Kap. 3 si uvedieme príklad z teórie riadenia, na riešenie ktorého je potrebné použitie maticových operácií netriviálnym spôsobom.

3 Lineárne kvadratická úloha

Dobrým príkladom využitia maticových operácií môže byť lineárne kvadratická (LQ) úloha, pozostávajúca zo sústavy lineárnych diferenciálnych rovníc (LDR) a z kvadratickej účelovej funkcie. Sústava LDR bude v tomto prípade stavová rovnica stavového modelu systému z Kap. 3.1, a kvadratickú účelovú funkcie si vysvetlíme bližšie v Kap. 3.2.

3.1 Stavový model

Stavový model systému oproti vstupno-výstupným modelom poskytuje niekoľko výhod. Stavovú reprezentáciu môžeme získať matematicko-fyzikálnou analýzou skutočného systému, alebo formálnymi matematickými úpravami z iných typov modelov. Lineárny, časovo invariantný systém v spojitom čase, sa dá vo všeobecnosti opísať stavovým modelom, ktorého tvar je

$$\dot{x}(t) = A_c x(t) + B_c u(t) \quad (3.1)$$

$$y(t) = C_c x(t) + D_c u(t), \quad (3.2)$$

kde $x(t)$ je n -rozmerný stavový vektor, ktorého prvkami sú stavové premenné (vnútorné premenné systému),

$$x(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ x_n(t) \end{bmatrix} \quad (3.3)$$

a vektor $\dot{x}(t)$ je deriváciou vektora $x(t)$ a teda platí

$$\dot{x}(t) = \begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \\ \vdots \\ \dot{x}_n(t) \end{bmatrix}, \quad (3.4)$$

kým $u(t)$ je m -rozmerný vektor vstupov a $y(t)$ je p -rozmerný vektor výstupov dané ako:

$$u(t) = \begin{bmatrix} u_1(t) \\ u_2(t) \\ \vdots \\ u_m(t) \end{bmatrix}, \quad y(t) = \begin{bmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_n(t) \end{bmatrix}. \quad (3.5)$$

Matica A_c je matica dynamického spojitého systému rozmeru $[n \times n]$, matica B_c je matica vstupu rozmeru $[n \times m]$, matica C_c je matica výstupu rozmeru $[p \times n]$, matica D_c je matica prevodu rozmeru $[p \times m]$ [17].

Pri synchrónnom vzorkovaní s periódou T bude diskretný stavový model zodpovedajúci spojitému modelu z Rov. (3.1)

$$x[(k+1)T] = Ax(kT) + Bu(kT), \quad (3.6)$$

$$y(kT) = Cx(kT) + Du(kT). \quad (3.7)$$

Pre normalizovanú periódou $T = 1$ vyjadríme tieto rovnice v jednoduchšom tvare

$$x(k+1) = Ax(k) + Bu(k) \quad (3.8)$$

$$y(k) = Cx(k) + Du(k), \quad (3.9)$$

kde matice C a D sú zhodné s maticami spojitého modelu C_c a D_c a pre matice A, B platí [10]:

$$A_c = \frac{1}{T} \ln A, \quad B_c = \frac{1}{T} (A - I)^{-1} (\ln A) B. \quad (3.10)$$

Ak chceme systém regulovať na základe diskretného stavového modelu, musí platiť

$$u(k) = -Kx(k), \quad (3.11)$$

kde K je zosilnenie stavového zákona, matica rozmeru $[m \times n]$.

3.2 Lineárne kvadratický regulátor

Máme teda Rov. (3.8), (3.9) a (3.11), predpokladajme ale, že použitie vstupov „niečo stojí“ a chceme nájsť také riadenie, ktoré bude najlepšie podľa nejakého kritéria. Toto kritérium predstavuje účelová funkcia. Účelová funkcia je umelo vytvorený numerický indikátor abstraktnej vlastnosti nejakého komplexného konceptu. Premieta viacero funkcií (trajektórií) premenných do reálneho čísla, ktoré by intuitívne vyjadriло „cenu“ zmien na vstupe, môže teda vyjadriť aké dobré riadenie máme [11]. LQ riadenie je vlastne optimalizačná úloha daná Rov. (3.8), (3.9) a (3.11), ktorá sa snaží minimalizovať účelovú funkciu danú vzťahom:

$$J_k = \frac{1}{2} \sum_{i=k}^{\infty} (x_i^T Q x_i + u_i^T R u_i), \quad (3.12)$$

kde Q - penalizačná matica stavov, R - penalizačná matica vstupov. Pre Q, R platí, že sú symetrické, kladne (semi-) definitné matice.

Pozrime sa na účelovú funkciu bližšie. Časť sumy $x_i^T Q x_i$ ako indikátor kvality riadenia vzniká ako druhá mocnina odchýlky riadenia. Bez ujmy na všeobecnosti predpokladajme, že máme nulovú referenciu, t.j. regulačnú úlohu. Druhú mocninu potrebujeme na to, aby bola každá odchýlka (kladná i záporná) zastúpená v sume, potom máme

$$x_i^T Q x_i = x_i^T C^T C x_i = y_i^T y_i, \quad (3.13)$$

z čoho

$$Q = C^T C. \quad (3.14)$$

Toto znamená, že namiesto stavov penalizujeme rovno výstupy, maticu Q však môžeme zvoliť aj inak. Stačí aby bola symetrická a kladne semi-definitná. Častou voľbou penalizačnej matice stavov je aj jednotková matica, t.j. $Q = I$, ktorá má taký význam, že rovnako penalizujeme všetky stavy. Ich minimalizácia a kvalita potom je rovnako dôležitá. Pri dodržaní matematických predpokladov vlastností matice Q môžeme robiť aj jemnejšie ladenie. Jednotlivé zložky na diagonále taktiež môžeme voľne meniť, pričom platí, že väčšie číslo znamená, že daný stav má v účelovej funkcii väčší význam.[11]

Druhým členom v sume sú vstupy rozšírené o penalizačnú maticu R ktorá opäť slúži na jemnejšie ladenie. Táto matica je tiež symetrická a kladne definitná (nemôže byť semi-definitná, lebo by to znamenalo elimináciu vplyvu riadenia). Všeobecne platí, že penalizačnú maticu vstupov si môžeme zvoliť tiež ľubovoľne.

Optimálne riadenie, ktoré minimalizuje účelovú funkciu (3.12) je dané ako:

$$u(k) = -Kx(k),$$

kde

$$K = (R + B^T P B)^{-1} B^T P A, \quad (3.15)$$

a maticu P dostávame riešením Lyapunovej rovnice.

$$P - (A + BK)^T P (A + BK) = Q + K^T R K. \quad (3.16)$$

Úloha výpočtu váhy P a LQ zosilnenia je potom vlastne riešením Rov. (3.16) a jedná sa o redukciu optimalizačného problému na riešenie diskkrétnej algebraickej Riccatiho rovnice (DARE), o ktorej si povieme niečo bližšie v nasledujúcej kapitole.

4 Riccatiho rovnica

Riccatiho rovnica je typ matematického problému, ktorý sa často vyskytuje pri riešení problémov optimálneho riadenia a optimálneho odhadu [11]. V uvedených príkladoch sa obmedzíme na algebraickú formu Riccatiho rovnice, určenej na riešenie dynamických problémov v ustálenom stave. Všeobecný tvar Riccatiho algebraickej rovnice pre diskretný čas je:

$$P - A^T P A + A^T P B (R + B^T P B)^{-1} B^T P A - Q = 0. \quad (4.1)$$

4.1 Algoritmy na riešenie diskretnéj algebraickej Riccatiho rovnice

Vyriešenie takejto rovnice ale vôbec nie je triviálne. Našťastie máme k dispozícii hneď niekoľko algoritmov, ktoré nám to umožnia, z ktorých si môžeme vyberať. Niekoľko z nich si aj uvedieme a zreplicujeme ich algoritmus vo forme funkcií v MATLABe. Uvedené príklady sú rozdelené do dvoch skupín, a to:

- iteratívne algoritmy
- hamiltonovské riešenia

Ako už aj názov naznačuje, iteratívne algoritmy „počítajú“ hodnotu matice váhovania stavov P v niekoľkých krokoch, naopak hamiltonovské riešenia patria medzi priame metódy, kde hľadanie riešenia Riccatiho rovnice je založené na nájdení vlastných čísel a vlastných vektorov hamiltonovskej matice¹ prislúchajúcej modelu.

4.1.1 Algoritmus „Jack Benny“

Najzákladnejší návod na riešenie nám ponúka hneď iteratívna metóda nazývaná autorom publikácie Implementation of Discrete Algebraic Riccati Equation Solver With Lapack in C++ Environment [21] „Jack Benny“² algoritmus. Tento algoritmus používa iteračný proces, v ktorom sa v každom kroku prepisujú matice P a K . Pre diskretný čas bude algoritmus vyzeráť nasledovne [21]:

¹Matica H je Hhmiltonovská práve vtedy, keď $(JH)^T = JH$, kde J je antisymetrická matica.

²Jack Benny je pseudonym amerického herca, komedistu Benjamína Kubelského. Metóda je pomenovaná po ňom pravdepodobne kvôli spôsobu, ako Jack Benny predstavoval nové postavy vo svojej relácii: zavolaním ich mena, akoby boli už dávno známi, napr. „Oh, Dennis...“. Obvyklá od poved bola „Áno pán Benny?“. Toto je veľmi podobné, ako vzájomné „oslovovanie sa“ matíc P a K v tomto algoritme

- Začínáme výberom ľubovoľnej matice zosilnenia K_0 , pre ktorú bude uzavretý systém $A + BK$ asymptoticky stabilný a ľubovoľnej matice P_0 . (Táto metóda, na rozdiel od ostatných metód uvedených v tejto práci požaduje aby bol uzavretý systém $A + BK$ asymptoticky stabilný, kým ďalšie metódy žiadajú len stabilizovateľnosť A, B .) Zvoľme si teda

$$\begin{aligned} K_0 &= 0 \\ P_0 &= I. \end{aligned} \tag{4.2}$$

- Následne Pre $k = 0, 1, 2, \dots$, nech je

$$P_k = (A - BK_k)^T P_k (A - BK_k) + Q + K_k^T R K_k \tag{4.3}$$

$$K_k = (R + B^T P_{k-1} B)^{-1} B^T P_{k-1} A, \tag{4.4}$$

potom $P = \lim_{k \rightarrow \infty} P_k$ rieši algebraickú Riccatiho Rov. (4.1).

Otázkou pri použití tohto algoritmu je počet iterácií, ktorý ale závisí od systému, yvoľme si preto prístup, kde počet iterácií je určený konvergenciou, nech: $|P_{k+1} - P_k| \leq \varepsilon$, kde $\varepsilon = 0.000000001$. Algoritmus potom v matlabe bude:

```
function [P K]=JackBenny(A,B,Q,R)
K=0; %Stabilne riesenie
Kpred=1;
i=0;
P=eye(1);
while abs(Kpred-K)>0.000000001*eye(length(K))
    Kpred=K;
    P=(A-B*K)'*P*(A-B*K)+K'*R*K+Q;
    K=inv(R+B'*P*B)*(B'*P*A);
end
    K=K;
    P=P;
end
```

4.1.2 Algoritmus zovšeobecnených vlastných čísel

Algoritmus zovšeobecnených vlastných čísel (Generalized Eigenproblem Algorithm, GEA) popísaný vo viacerých prácach, napríklad [21] a [20], sa radí medzi hamiltonovské riešenia, hamiltonovská matica teda zohráva dôležitú úlohu v tomto algoritme.

- Pre spojitý čas sa zostrojí hamiltonovská matica H ako:

$$H = \begin{bmatrix} A^{-1} & -A^{-1}BR^{-1}B \\ -Q & -A \end{bmatrix}. \tag{4.5}$$

- V druhom kroku hľadáme maticu vlastných vektorov W , pre ktorú platí,

$$W^{-1}HW = \begin{bmatrix} \Lambda_s & 0 \\ 0 & \Lambda_u \end{bmatrix},$$

kde Λ_s je Jordanovská bunka³, ktorej reálna časť vlastných čísel je záporná, a Λ_u je Jordanovská bunka, ktorej reálna časť vlastných čísel je kladná.

Toto je ekvivalentné s hľadaním zovšeobecnených vlastných čísel matíc L a M , ktoré majú tvar

$$L = \begin{bmatrix} I & BR^{-1}B \\ 0 & -1 \end{bmatrix}, \quad M = \begin{bmatrix} A & 0 \\ -Q & I \end{bmatrix}.$$

- Na koniec ak rozdelíme maticu W na štyri bloky ako

$$W = \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix},$$

potom matica $P = W_{21}W_{11}^{-1}$ rieši Rov. (4.1).

Implementácia v MATLABe potom bude vyzeráť nasledovne:

```
function [P K]=Generalized(A,B,Q,R)
    %inicializacia matic L a M
    G=B*inv(R)*B';
    l=length(A);
    L=[eye(l)    G;
        zeros(l) A'];
    M=[A zeros(l);
        -Q eye(l)];
    %najdenie zovseobecnenych vlastnych cisel a
    vektorov L a M
    [W,D]=eig(L,M);
    %indexy na rozdelenie matice W
    i=length(W)/2;
    j=length(W);
    %vypocet P a K
    P=real(W((i+1):j,1:i)*inv(W(1:i,1:i)));
    K=inv(R+B'*P*B)*B'*P*A;
end
```

³Jordanovská bunka je matica, na ktorej diagonále sa nachádzajú jej vlastné čísla, a nad nimi sú jednotky a všade inde nuly.

4.1.3 Schurova metóda

Schurova metóda tiež popísaná v práci Y. Zhenga [21] je veľmi podobná problému všeobecných vlastných čísel, hlavným rozdielom je, že namiesto vektorov vlastných čísel, sa pri dekompozícii hamiltonovskej matice použije Schurov rozklad.

- Pri tomto algoritme hamiltonovská matica pre diskretný čas sa skonštruuje ako:

$$H = \begin{bmatrix} A + BR^{-1}B^T(A^{-1})^TQ & BR^{-1}B^T(A^{-1})^T \\ -(A^{-1})^TQ & (A^{-1})^T \end{bmatrix}. \quad (4.6)$$

- Hľadáme maticu W , pre ktorú platí:

$$W^{-1}HW = S = \begin{bmatrix} S_{11} & S_{12} \\ 0 & S_{22} \end{bmatrix},$$

kde matica S je kvázi-horná trojuholníková matica a všetky reálne časti vlastných čísel submatice S_{11} sú záporné a všetky reálne časti vlastných čísel submatice S_{22} sú kladné.

- Potom ak rozdelíme maticu W na štyri bloky ako

$$W = \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix}.$$

Matica $P = W_{21}W_{11}^{-1}$ potom rieši Rov.(4.1).

Algoritmus v MATLABe bude:

```
function [P K]=Schur(A,B,Q,R)
    %definicia hamiltonovskej matice Z
    H=[A+B*inv(R)*B'*inv(A)'+Q      -B*inv(R)*B
        '*inv(A)';
        -inv(A)'+Q                      inv(A)
        '];
    %Schurov rozklad
    [W,T]=schur(H);
    %indexy na rozdelenie matice W
    i=length(W)/2;
    j=length(W);
    %vypocet P a K
    P=Ws((i+1):j,1:i)*inv(Ws(1:i,1:i));
    K=inv(R+B'*P*B)*B'*P*A;
end
```

4.1.4 Nerekurzívna metóda

V prácach Vaughana [19] a Adama a Assimakisa [5] nájdeme návod na ďalší algoritmus - na nerekurzívnu metódu riešenia Rov. (4.1).

- Hamiltonovskú maticu skonštruujeme ako

$$H = \begin{bmatrix} A + BR^{-1}B^T(A^{-1})^TQ & -BR^{-1}B^T(A^{-1})^T \\ -(A^{-1})^TQ & (A^{-1})^T \end{bmatrix}. \quad (4.7)$$

- Tak, ako aj pri probléme zovšeobecnených vlastných čísel hľadáme maticu vlastných vektorov W . Po rozdelení matice W na štyri bloky ako

$$W = \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix}$$

opäť platí, že reálna časť $P = W_{21}W_{11}^{-1}$ rieši Rov. (4.1)

V MATLABe máme:

```
function [P K]=Nonrec(A,B,Q,R)
    %definicia hamiltonovskej matice Z
    Rc=B*inv(R)*B';
    H=[(A+Rc*(inv(A')))-Rc*inv(A');
        -inv(A')*Q inv(A')];
    %najdenie vlastnych cisel inv(H)
    [W,D]=eig(inv(H));
    %indexy na rozdelenie matice W
    i=length(W)/2;
    j=length(W);
    %vypocet P a K
    P=real(W((i+1):j,1:i)*inv(W(1:i,1:i)));
    K=real(inv(R+B'*P*B)*B'*P*A);
end
```

Nerekurzívna metóda v MATLABe

4.2 Porovnanie algoritmov na riešenie Riccatiho rovnice

Majme diskretný model

$$A = \begin{bmatrix} 0.8631 & 0.0095 \\ -26.7180 & 0.8577 \end{bmatrix}, \quad B = 1E-4 \begin{bmatrix} 0.0027 \\ 0.5344 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 0 \end{bmatrix} \text{ a } D = 0 \quad (4.8)$$

a nech sú penalizačné matice stavov a vstupov

$$Q = C^T C, \quad R = 1E-4 \quad (4.9)$$

	Matica váhovania P	Matica zosilnenia K
MATLAB <code>dare()</code>	$\begin{bmatrix} 89.5023 & 0.0170 \\ 0.0170 & 0.0317 \end{bmatrix}$	$\begin{bmatrix} -0.2342 & 0.0170 \end{bmatrix}$
Jack Benny	$\begin{bmatrix} 89.5022 & 0.0170 \\ 0.0170 & 0.0317 \end{bmatrix}$	$\begin{bmatrix} -0.2342 & 0.0170 \end{bmatrix}$
GEA	$\begin{bmatrix} 89.5023 & 0.0170 \\ 0.0170 & 0.0317 \end{bmatrix}$	$\begin{bmatrix} -0.2342 & 0.0170 \end{bmatrix}$
Schurova metóda	$\begin{bmatrix} 89.5023 & 0.0170 \\ 0.0170 & 0.0317 \end{bmatrix}$	$\begin{bmatrix} -0.2342 & 0.0170 \end{bmatrix}$
Nerekurzívna metóda	$\begin{bmatrix} 89.7315 & 0.0158 \\ 0.0181 & 0.0318 \end{bmatrix}$	$\begin{bmatrix} -0.2342 & 0.0170 \end{bmatrix}$

Tabuľka 4.1: Výsledky v MATLABe

	Jack Benny	GEA	Schurova metóda	Nerekurzívna metóda
$\max P_{ij}^{ref} - P_{ij}^{func} $	0.0001	0.0000	0.0000	0.2292

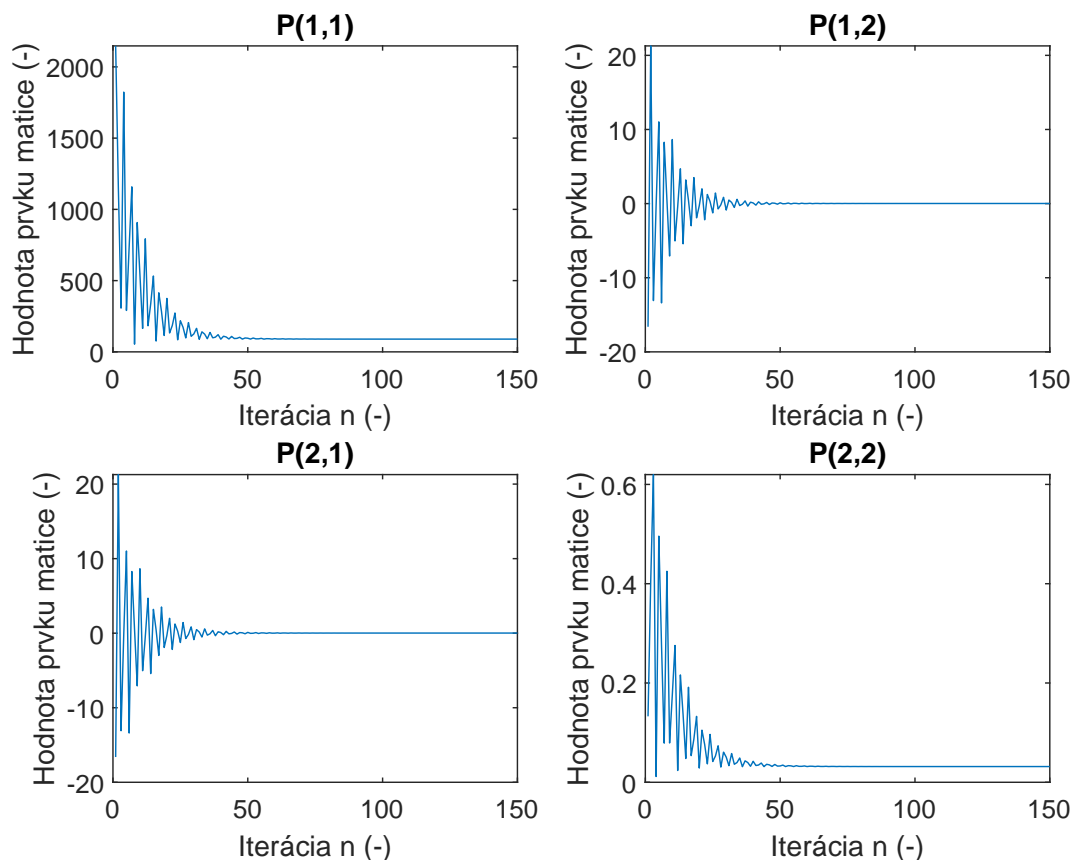
Tabuľka 4.2: Rozdiely medzi prvkami matíc P^{ref} a P^{func}

Riešme Riccatiho rovnicu pre tieto vstupy pomocou vyššie uvedených algoritmov a zabudovanej funkcie z Matlabu `dare()` dostávame:

Po porovnaní výsledkov, ktoré sme dostali použitím vlastných funkcií s výsledkom funkcie `dare` v MATLABe určenej na riešenie diskkrétnej algebraickej Riccatiho rovnice dostávame Tab. 4.2, ktorá obsahuje najväčšie rozdiely prvkov matice počítanej pomocou `dare`, ozn. ako P^{ref} a vlastnými funkciami, ozn. ako P^{func} . Vidíme, že najnepresnejším algoritmom je nerekurzívna metóda. Druhý najnepresnejší výsledok sme dostali iteratívnym algoritmom. Treba však pripomenúť, že termináciu algoritmu sme podmienili konvergenciou - tento výsledok sme dostali po 2353 iteráciách. Konvergenciu prvkov matice P sme znázornili vykreslením každej dvadsiatej iterácie na Obr. 4.1. Pri algoritme „Jack Benny“ by sme poznamenali, že stabilita a konvergencia algoritmu je závislá od modelu, ktorý do algoritmu vstupuje (hlavne od podmienenosti matice systému).

Aj pri porovnaní algoritmov na riešenie Riccatiho rovnice by do veľkej miery pomohlo, keby sme vedeli zistiť, koľko pamäte jednotlivé algoritmy spotrebujú, a aký čas je potrebný na výpočet. Žiaľ tu narážame na ťažkosti vyplývajúce z prostredia MATLAB a zo skutočnosti, že sa výpočty realizujú na zariadení, na ktorom nedokážeme ovplyvniť všetky procesy súčasne prebiehajúce na CPU — na našom počítači.

Meranie času v matlabe sa dá realizovať párovými príkazmi `tic()` a `toc()`,



Obr. 4.1: Konvergencia prvkov matice P

alebo využitím MATLAB Profileru. V oboch prípadoch si však treba uvedomiť, že čas výpočtu závisí od konkrétneho hardvéru a tiež od procesov a aplikácií ktoré sú spustené súčasne s meraným výpočtom.

Meranie použitej pamäte v MATLABe je už trochu obtiažnejšie. Vo verzií MATLABu R2008 existovali funkcie na meranie použitej pamäte podobné funkciám `tic()` a `toc()`, a to `mtic()` a `mtoc()`. Táto užitočná pomôcka sa však do novších verzií nedostala, namiesto toho nami používaná verzia MATLABu R2018b ponúka nezdokumentovanú možnosť rozšírenia MATLAB Profileru o sledovanie pamäte: Alokovaná pamäť, uvoľnená pamäť, vlastná pamäť a maximálna využitá pamäť.

Vykonali sme s použitím `for()` cyklu výpočet riešenia Riccatiho rovnice a zosilnenia pre ten istý systém uvedenými algoritmami, následne sme tieto výpočty vykonali aj bez `for` cyklu, raz. Výsledky sú uvedené v Tab. 4.3.

calls	„JackBenny“		GEA		Schurova m.		Nerekurzívna m.	
	100	1	100	1	100	1	100	1
Celkový čas[s]	2,732	0,063	0,009	0,005	0,011	0,004	0,011	0,006
Vlastný čas*[s]	2,732	0,063	0,009	0,005	0,011	0,004	0,011	0,006
Alokovaná pamäť[Kb]	284	64	0	0	0	0	0	0
Uvoľnená pamäť[Kb]		64	0	0	0	0	0	0
Vlastná pamäť[Kb]	0	0	0	0	0	0	0	0
Maximálna využitá pamäť[Kb]	64	16	0	0	0	0	0	0

Tabuľka 4.3: Porovnanie algoritmov v MATLABe pomocou príkazu profiler

Keď sa teda pozrieme na tabuľku 4.3 zistíme niekoľko zaujímavostí:

- meranie použitej pamäte pomocou Profileru nevyhovuje našim potrebám, keďže to vyzerá, ako keby hamiltonovské solvre nevyužívali počas výpočtu žiadnu pamäť;
- rozdiel trvania výpočtov s for cyklom a bez neho je príliš malý na to, aby sme nepredpokladali, že MatLab rozmýšľa aj za nás a tieto výpočty optimalizuje;
- jediné tvrdenie, ktoré môžeme s istotou vysloviť na základe týchto výstupov je, že Jack Benny algoritmus je najpomalší a počas výpočtov najnáročnejší na využitie pamäte.

Toto nám na porovnanie uvedených algoritmov na riešenie Riccatiho rovnice nestačí, môžeme však zhrnúť výhody a nevýhody jednotlivých algoritmov. Najväčšou výhodou algoritmu „Jack Benny“ je, že počas výpočtov používa len elementárne maticové operácie, na druhej strane čas potrebný na výpočet (ktorý závisí od počtu iterácií, čo je zas závislé od systému) je väčší, a tento algoritmus nešetrí ani pamäťou.

Naopak hamiltonovské riešenia (vrátane menej presnej nerekurzívnej metódy) sú rýchle a spoľahlivé, vhodné aj na on-line počítanie LQ zosilnenia. Napriek tomu, v ďalších uvahách sa musíme obmedziť na algoritmus Jack Benny, kvôli chýbajúcej softvérovej podpore pokročilejších maticových operácií, ako nájdenie vlastných čísel, vlastných vektorov, Schurov rozklad, a pod. Nevyhnutnosť týchto operácií sme si overili aj exportom C++ programu z MATLABu, kde zložitejšie operácie (nájdenie vlastných čísel) sú tiež riešené používaním knižníc známych v jazyku C++, nie však ľahko implementovateľné v Arduino IDE.

5 Algoritmus „Jack Benny“ na mikroovládačoch s architektúrou AVR

V tejto kapitole si ukážeme výpočet váhovania koncového stavu pomocou algoritmu „Jack Benny“, za použitia knižníc MatrixMath a BLA. Vypočítajme predošlý príklad (4.8) a (4.9) na mikropočítači AVR. Pre jednoduchosť ale použijeme namiesto `while()` cyklu `for()` pre iterácie (ušetříme si tým výpočet rozdielov medzi dvomi iteráciami a porovnanie s nami zvolenou presnosťou).

Na prvý pohľad vidíme rozdiely v používaní knižníc. Nakoľko knižnica MatrixMath je zväčša len zbierkou algoritmov vo forme `void` funkcií, kde pre každú operáciu do funkcie vstupuje ako argument aj očakávaný výstup, navyše sa rozmery matíc musia kontrolovať manuálne, musíme vytvárať dočasné premenné a dostávame pomerne neprehľadný kód:

```
//iteracia P s pouzitim kniznice MatrixMath
Matrix.Multiply((mtx_type*)B, (mtx_type*)K, 2, 1, 2, (
    mtx_type*)Temp1); //B[Nx1]*K[1*N]
Matrix.Subtract((mtx_type*)A, (mtx_type*)Temp1, 2, 2, (
    mtx_type*)Temp2); //A-B[Nx1]*K[1*N]
Matrix.Transpose((mtx_type*)Temp2, 2, 2, (mtx_type*)
    Temp3); //(A-B[Nx1]*K[1*N])'
Matrix.Multiply((mtx_type*)Temp3, (mtx_type*)P, 2, 2,
    2, (mtx_type*)Temp4); //(A-B[Nx1]*K[1*N])'*P
Matrix.Multiply((mtx_type*)Temp4, (mtx_type*)Temp2, 2,
    2, 2, (mtx_type*)Temp5); //(A-B[Nx1]*K[1*N])'*P(A-B[
    Nx1]*K[1*N])
//zjednodusenie vyrazu K'RK, ktore sme si mohli dovolit
    len vďaka tomu, že R je v tomto prípade skalar
Matrix.Multiply((mtx_type*)K, (mtx_type*)K, 2, 1, 2, (
    mtx_type*)Temp6);
Matrix.Scale((mtx_type*)Temp6, 2, 2, R);
Matrix.Add((mtx_type*)Temp5, (mtx_type*)Temp6, 2, 2,
    (mtx_type*)Temp7); //(A-B[Nx1]*K[1*N])'*P(A-B[Nx1]*K
    [1*N])+K'RK
Matrix.Add((mtx_type*)Temp7, (mtx_type*)Q, 2, 2, (
    mtx_type*)Temp8); //(A-B[Nx1]*K[1*N])'*P(A-B[Nx1]*K
    [1*N])+K'RK+Q
Matrix.Copy((mtx_type*)Temp8, N, N, (mtx_type*)P);
```

	Matica váhovania P	Matica zosilnenia K
MATLAB	$\begin{bmatrix} 89.50 & 0.02 \\ 0.02 & 0.03 \end{bmatrix}$	$\begin{bmatrix} -0.23 & 0.02 \end{bmatrix}$
MatrixMath	$\begin{bmatrix} 85.03 & 0.02 \\ 0.02 & 0.03 \end{bmatrix}$	$\begin{bmatrix} -0.22 & 0.02 \end{bmatrix}$
BLA	$\begin{bmatrix} 85.03 & 0.02 \\ 0.02 & 0.03 \end{bmatrix}$	$\begin{bmatrix} -0.22 & 0.02 \end{bmatrix}$

Tabuľka 5.1: Výsledky v MATLABe

Oproti tomu tento istý výpočet vieme zapísať pomocou knižnice BLA v jednom riadku.

```
//iteracia P pouzitim BLA
P=(~(A-B*K)*P)*(A-B*K)+Q+~K*R*K;
```

Pri výpočte zosilnenia K je opäť riešenie pomocou knižnice BLA elegantnejšie, aj keď v tomto prípade tiež musíme deklarovať jednu dočasnú premennú `Inv`:

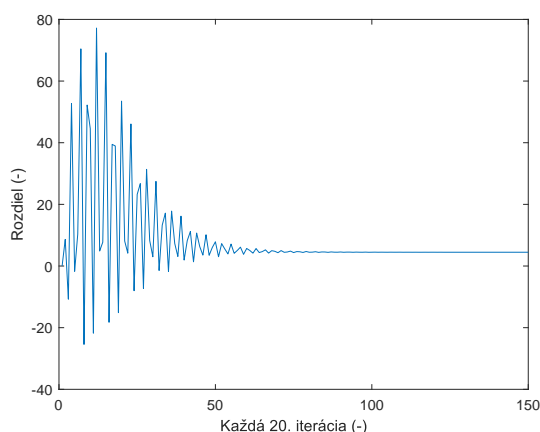
```
//iteracia K s pouzitim BLA
Inv=(R+(~B*P)*B); //docasna premenna
K=Invert(Inv)*((~B*P)*A);
```

To isté použitím knižnice `MatrixMath` opäť vyzerá neprehľadne a náročne:

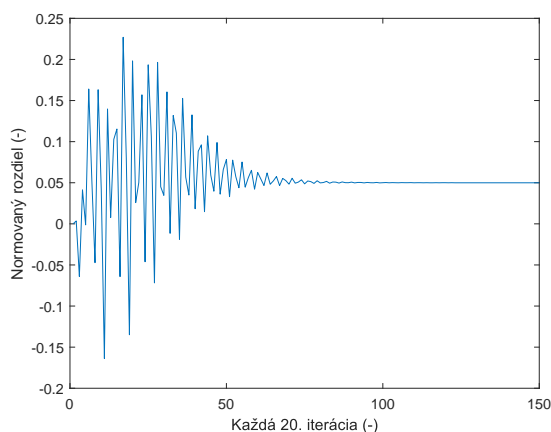
```
//iteracia K s pouzitim MatrixMath
Matrix.Multiply((mtx_type*)B, (mtx_type*)P, 1, 2, 2, (
    mtx_type*)Temp21);//
Matrix.Multiply((mtx_type*)Temp21, (mtx_type*)B, 1, 2,
    1, (mtx_type*)Temp22);//
f1=Temp22[0][0]+R;
Matrix.Multiply((mtx_type*)Temp21, (mtx_type*)A, 1, 2,
    2, (mtx_type*)Temp23);//
Matrix.Scale((mtx_type*)Temp23, 1, 2, 1.00/f1);//
Matrix.Copy((mtx_type*)Temp23, 1, 2, (mtx_type*)K);
```

Na výpočet matice P v MATLABe sme potrebovali o niečo menej ako 3000 iterácií, zvolme si preto počet cyklov `int cycle=3000`; a vypočítajme maticu váhovania P a maticu LQ zosilnenia K pomocou oboch knižníc. Po porovnaní získaných výsledkov v Tab. 5.1 vidíme, že po zaokrúhlení výsledkov na dve desatinné miesta je najväčší rozdiel iba v prvku matice $P(1,1)$. Medzi výsledkami na AVR nie je rozdiel, bezodhľadu na to, akú knižnicu použijeme, preto sa v nasledujúcich úvahách obmedzíme na knižnicu BLA. Porovnajme si teda každú dvadsiatu iteráciu vypočítanu v MATLABe pomocou vlastnej funkcie `JackBenny()` a každú dvadsiatu iteráciu počítanu s použitím knižnice BLA. Toto porovnanie vidíme na Obr. 5.1. Z tohto grafu sa zdá, že sú rozdiely v jednotlivých iteráciách, hlavne v prvých

krokoch, dosť veľké. Treba si však uvedomiť, že v prvých iteráciách sú aj prvky matice veľké čísla. Na Obr. 5.2 teda vidíme rozdiely predelené očakávanými hodnotami z MATLABu. Tieto rozdiely sú spôsobené zaokrúhľovaním na AVR pri prvých iteráciách, kde hodnota $P(1,1)$ oscilovala medzi hodnotami rádovo desiatok a tisícok. Nestabilita výpočtu je teda ovplyvnená podmienenosťou matice A , ktorá v tomto prípade dosahuje hodnotu $\text{cond}(A)=719$. To, či takáto presnosť je postačujúca v praxi, závisí od konkrétneho systému. V ďalšom kroku si ukážeme parametrizáciu algoritmu, resp. si z nej vytvoríme funkciu, ktorú použijeme na inom, ľahšom príklade.



Obr. 5.1: Rozdiel medzi MATLABom a BLA



Obr. 5.2: Rozdiel medzi MATLABom a BLA predelený očakávanou hodnotou

5.1 Funkcia na výpočet LQ zosilnenia

Pri tvorbe funkcie, ktorej výstupom má byť celé pole, špeciálne matica s prvkami typu `float`, do ktorej navyše ako parametre musia vstupovať aj rozmery vstupných matíc, je najintuitívnejšie zvoliť prístup známy aj zo samotnej knižnice BLA: tvorbu

šablón, templateov. Zaujímavosťou takýchto funkcií je, že ako argumenty do nich vstupujú nie len samotné vstupné premenné, ale aj ich typy. Toto zjednodušuje následné použitie takejto funkcie, za cenu trochu obtiažnejšieho programovania pri jej tvorbe.

V LQ riadení sa nevyužíva matica váhovania P , ale matica zosilnenia K , ktorú tiež dostávame z algoritmu „Jack Benny“. Určíme si teda túto maticu ako výstup. Vstupné argumenty pri volaní funkcie by mali byť matice A, B, Q, R a môžeme si k nim pridať aj počet cyklov. Pre efektívnejší výpočet si zahrňme aj matice P a K medzi vstupné argumenty. Ušetríme tým alokáciu pamäte v rámci funkcie a uľahčíme vyčítanie výsledku po výpočte. Toto tiež znamená, že na konci výpočtu budeme mať obe matice (P, K) určené, a rozhodnutie označenia matice zosilnenia ako výstupu je len estetická úprava. Ak chceme, aby sa táto funkcia dala zavolať príkazom `K=gain(A,B,Q,R,P,K,int cycle)`, definujeme si šablónu ako

```
//definicia sablony
template< int m, int n, int p, class Pmat, class Amat,
        class Bmat, class Qmat, class Rmat, class Kmat>
//ocakavany vystup funkcie, nazov, vstupne argumenty
Matrix<m,n,Kmat> &gain(Matrix<n,n,Amat> &A,Matrix<n,m,
        Bmat> &B, Matrix<p,p,Qmat> &Q, Matrix<m,m,Rmat> &R,
        Matrix<n,n,Pmat> &P, Matrix<m,n,Kmat> &K, int cycle)
```

Parametre m, n, p , ako rozmery jednotlivých matíc vyplývajú zo stavovej reprezentácie systému uvedenej v Kap. 3 a sú spolu s triedami `Pmat`, `Amat`, `Bmat`,

`Qmat`, `Rmat`, `Kmat` až po zavolanie funkcie a kompiláciu v podstate len premenné, ktoré kompilátor doplní podľa vstupných argumentov funkcie. Alokácia výstupných premenných `Matrix<n,n,Pmat> &P`, `Matrix<m,n,Kmat> &K` a ich používanie ako vstupné argumenty — ako sme už spomínali — zabezpečí spoľahlivejší a rýchlejší výpočet. Vstupný argument `cycle` umožní užívateľovi rozhodnúť o počte iterácií, tento argument sa dá definovať priamo pri zavolaní funkcie, nie je nutné ho teda vkladať do definície šablóny. Samotná funkcia sa potom skladá z inicializácie premenných K, P a dočasnej premennej Inv na výpočet iterácie K ,

```
Matrix<m,m> Inv; //docasna premenna k vypoctu K

//Inicializacia K, ako lubovolneho stabilneho riesenia
systemu
for(int i = 0; i < m; i++){
    for(int j = 0; j < n; j++){
        K(i,j) = 0.00000000;
    }
}

//inicializacia P=I
for(int i = 0; i < n; i++){

    for (int j = 0; j < n; j++) {
```

```

        if (i == j) {
            P(i, j) = 1.00;
        }
        else {
            P(i, j) = 0.00;
        }
    }
}
} //koniec inicializacie P
a zo samotnej iterácie K a P:
for(int i=0; i<cycle ;i++){ //cyklus vykonavajuci
    iteracie
//iteracia P
P=(~(A-B*K)*P)*(A-B*K)+Q+~K*R*K;

//iteracia K
Inv=(R+(~B*P)*B);
K=Invert(Inv)*((~B*P)*A);
} //koniec cyklu

return P;

```

5.2 Príklad použitia funkcie

Majme jednoduchú regulačnú úlohu. Predstavme si satelit (Obr. 5.3), vychýlený v jednej osi o uhol θ z referenčnej polohy (bez ujmy na všeobecnosti nech je táto referenčná poloha nulová). Ak neuvažujeme poruchový stav, rotačnú dynamiku satelitu môžeme napísať ako:

$$\begin{aligned}
 I\ddot{\theta} &= M_C \\
 u &= \frac{M_C}{I},
 \end{aligned} \tag{5.1}$$

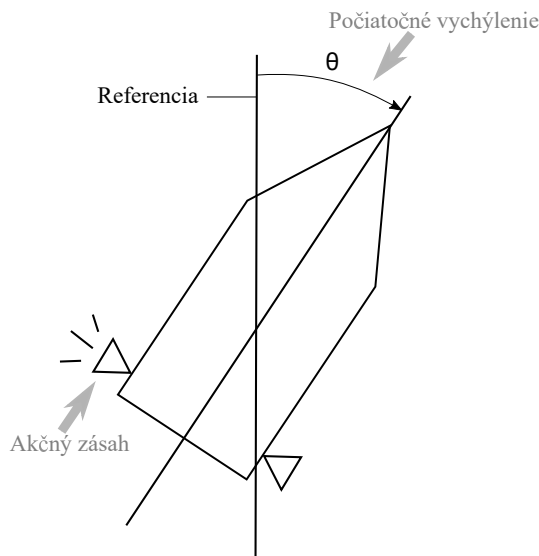
kde I je moment zotrvačnosti, $\ddot{\theta}$ uhlové zrýchlenie, M_C riadiaci moment a u akčný zásah. Po dosadení a úpravách máme:

$$\ddot{\theta} = u. \tag{5.2}$$

Spojité stavová reprezentácia tohto systému je:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}}_{A_c} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \underbrace{\begin{bmatrix} 0 \\ 1 \end{bmatrix}}_{B_c} u \tag{5.3}$$

$$y = \underbrace{\begin{bmatrix} 1 & 0 \end{bmatrix}}_{C_c} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \tag{5.4}$$



Obr. 5.3: Príklad satelitu

kde $y = \theta$ je výstup a $x_1 = \theta$, $x_2 = \dot{\theta}$ sú stavy. Zvoľme si vzorkovanie ako

$$T = 1.$$

Pomocou príkazu `c2dm()` v MATLABe si diskretizujeme tento model a dostávame (5.5) a (5.6)

$$A = \begin{bmatrix} 1.00 & 1.00 \\ 0.00 & 1.00 \end{bmatrix}, \quad B = \begin{bmatrix} 0.50 \\ 1.00 \end{bmatrix}, \quad C = C_c, \quad D = D_c \quad (5.5)$$

a nech sú penalizačné matice stavov a vstupov

$$Q = C^T C, \quad R = 1.00. \quad (5.6)$$

Poznamenajme, že tento príklad je omnoho jednoduchší, ako príklad daný Rov. (4.8) a (4.9) a platí preň `cond(A) = 2.6`. Po vypočítaní príkladu (5.5), (5.6) v MATLABe aj pomocou vlastnej funkcie v C++ `gain()` sme po zaokrúhlení na dve desatinné miesta dostali rovnaké výsledky.

$$K = \begin{bmatrix} 0.50 & 1.00 \end{bmatrix}, \quad P = \begin{bmatrix} 2.00 & 1.00 \\ 1.00 & 1.50 \end{bmatrix},$$

Funkcia na AVR pri výpočte tohto príkladu zabrala 56 236 bajtov pamäte a program zbehol v priemere zo 100 zavolaní za 12 299,96 μs , čo zodpovedá zhruba 1500 sčítavaniam dvoch premenných typu `float` do tretej, aj s uložením na RAM, resp. 100 výpočtom `sin()` pre akýkoľvek dátový typ podľa Tab. 1.2 a 1.3.

Spravme si simuláciu riadenia tohto systému v uzavretej slučke, pre $t = 0, 1, 2, \dots, 10$. Nech je počiatočný stav:

$$x(0) = \begin{bmatrix} 1.00 \\ 1.00 \end{bmatrix}, \quad (5.7)$$

potom po dosadení do 3.8 a 3.9 môžeme písať v MATLABe:

```

\%deklaracia matic
    A=[1 1;0 1];
    B=[0;1];
    C=[1 0];
    Q=C'*C;
    R=1;
\%vypocet matice zosilnenia K
    K=dlqr(A,B,Q,R);
\%simulacia
    x0=[1;1]; \%pociatocny stav
    X1=x0;
    t=0:1:10; \%cas
    N=length(t);\% pocet krokov simulacie
\% vypocet simulacie
    for k=1:N
        U(k)=-K*X1(:,k);
        X1(:,k+1)=A*X1(:,k)+B*U(k);
        Y1(k)=C*X1(:,k);
    end

```

a v C++ máme:

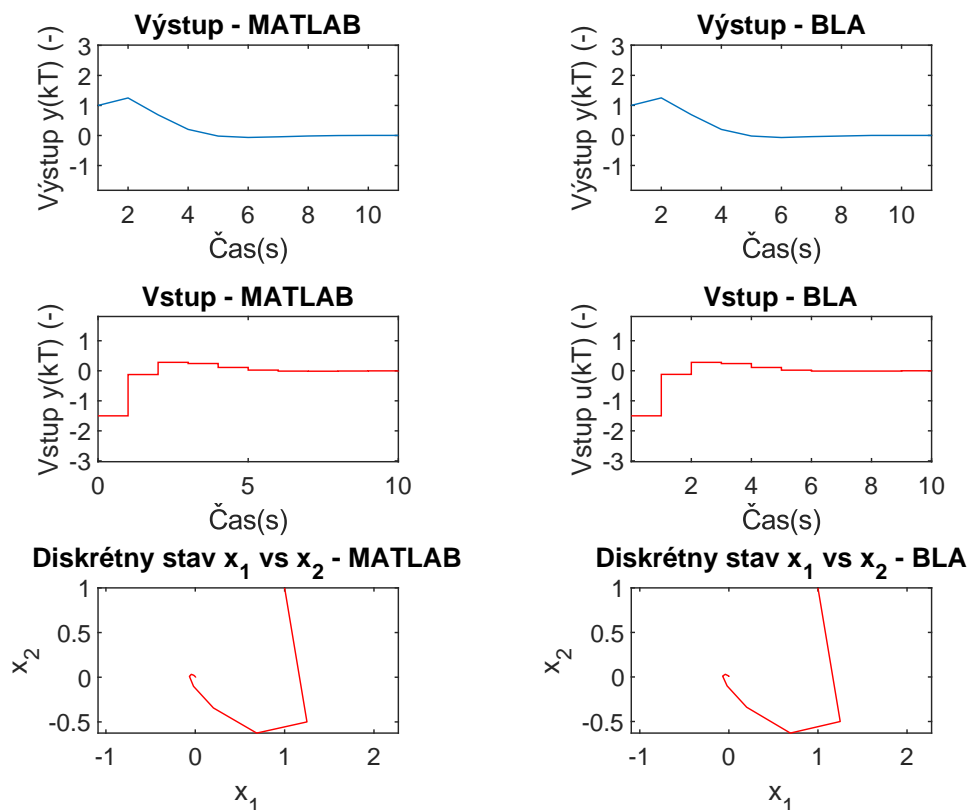
```

//deklaracia matic
    Matrix<2,2> A;
    A<<      1.00,    1.00,
            0.00,    1.00;
    Matrix<2,1> B;
    B<<      0.50,
            1.00;
    Matrix<1,2> C;
    C<<      1.00, 0.00;
    Matrix<2,2> Q=~C*C;
    Matrix<1,1> R=1;
//vypocet matice zosilnenia K pomocou vlastnej funkcie
    gain()
    K=gain(A,B,Q,R,P,K,15); //15 iteracii
//simulacia - pociatocny stav
    X1<<1.00,
        1.00;
//vypocet simulacie
    for(int t=0;t<11;t++){
        U=-K*X1;
        Y1=C*X1;
        X1=A*X1+B*U;
    }

```

Dospeli sme opäť k rovnakým výsledkom, ktoré sú zobrazené na Obr. 5.4

Môžeme teda skonštatovať, že nami navrhnutá funkcia výpočtu LQ zosilnenia



Obr. 5.4: Simulácia v MATLABe a v BLA

funguje, jej presnosť je však obmedzená architektúrou AVR a závisí od podmienosti matice systému. Okrem samotnej presnosti, aj to, či je dosiahnutá presnosť „dostatočná“ závisí od konkrétného systému. Zaujímavým predmetom ďalšieho prieskumu by mohlo byť testovanie „dostatočnej“ presnosti v praktických príkladoch. Môžeme však vyhlásiť, že napriek ťažkostiam a obmedzeniam na AVR, pomocou knižnice BLA sa dajú bezproblémovo používať maticové operácie na týchto mikroovládačoch, dokonca aj pri zložitejších algoritmoch — pokiaľ sa tieto algoritmy opierajú len o elementárne maticové operácie. Dalším príjemným zjednodušením maticových výpočtov by bolo, keby knižnica BLA obsahovala aj často sa vyskytujúce zložitejšie úkony, ako sú rôzne rozklady a transformácie.

6 Záver

V tejto práci sme uviedli stručný prehľad dostupných riešení pre aplikáciu maticových operácií na programovateľných mikroovládačoch AVR. Ako najefektívnejšie nám prišlo použiť niektorú z existujúcich knižníc. Otestovali sme štyri dostupné knižnice:

- MatrixMath,
- Matrix,
- BasicLinearAlgebra (BLA),
- Eigen,

z ktorých najvýhodnejšiou sa ukázala knižnica BLA, napriek niektorým užitočným vlastnostiam knižnice MatrixMath. Knižnica MatrixMath zaostávala hlavne v užívateľskej prívetivosti a jednoduchosti, resp. prehľadnosti používania. Ďalšie dve knižnice sme zavrhlí hneď po prvých testoch. Knižnica Matrix sa ukázala ako slabší, nie dostatočne rozvinutý ekvivalent knižníc BLA a MatrixMath. Prekvapivo ani široko odporúčaná knižnica Eigen sa neukázala dostatočná, napriek mnohým funkciám a elegantnému používaniu ktoré používa. Jej najväčšia nevýhoda spočíva práve v slabej implementácii na AVR, hlavne pri využívaní pamäťových jednotiek mikroovládača, čo sa prejavilo už pri výkone elementárnych operácií s maticami s rozmermi väčšími ako 4×4 . Po vybratí knižníc MatrixMath a BLA sme uviedli príklad problému, na ktorého riešenie je potrebný zložitejší algoritmus - diskretnú algebraickú Riccatiho rovnicu, ktorej potrebnosť v teórii riadenia vyplýva z LQ riadenia. Uviedli sme dva prístupy k riešeniu diskretnéj algebraickej Riccatiho rovnice:

- iteratívne algoritmy a
- hamiltonovské riešenia.

Bližšie sme skúmali štyri algoritmy (1 iteratívny a 3 hamiltonovské) z hľadiska výkonnosti. Napriek tomu, že z hľadiska numerických výpočtov najnevýhodnejším algoritmom, ako bolo aj očakávateľné, sa ukázal iteratívny algoritmus, nazývaný „Jack Benny“, kvôli ohrozeniam vybraných knižníc do ďalších úvah sme boli nútení zahrnúť práve tento spôsob riešenia. V poslednej kapitole sme zreprodukovali algoritmus za použitia oboch knižníc, MatrixMath a BLA. Následne sme za použitia knižnice BLA tento algoritmus parametrizovali a vytvorili z neho funkciu `gain()` na výpočet LQ zosilnenia K . Túto funkciu sme otestovali na jednoduchej regulačnej úlohe, v ktorej nielenže sme vypočítali LQ zosilnenia, ale aj simulovali riešeny

system. Ďalším krokom by mohla byť hlbšia analýza možností vylepšenia použiteľnosti uvedených knižníc na AVR, ako napr. vytvorenie lepšieho rozhrania pre použitie knižnice Eigen, zlepšenie zobrazenia matíc v knižnici MatrixMath, príp. rozšírenie knižnice BLA o zložitejšie výpočty, ako napríklad nájdenie vlastných čísel a vlastných vektorov matice alebo Schurov rozklad, ktoré by umožnili jednoduchú implementáciu aj zložitejších algoritmov, ako sú vo všeobecnosti iteratívne prístupy. Užítok tejto práce by tiež mohlo byť jej využitie ako základu pre tvorbu knižnice lineárne kvadratického riadenia pre Arduino, hlavne na didaktické účely.

Literatúra

- [1] Armadillo, c++ library for linear algebra scientific computing. Online. [cit. 2019-02-17] <http://arma.sourceforge.net/>.
- [2] Blas (basic linear algebra subprograms). Online. [cit. 2019-02-17] <http://www.netlib.org/blas/>.
- [3] blaze. Online. [cit. 2019-02-17]. <https://bitbucket.org/blaze-lib/blaze>.
- [4] Lapack—linear algebra package. Online. [cit. 2019-02-17]. <http://www.netlib.org/lapack/>.
- [5] M. Adam and N. Assimakis. Nonrecursive algebraic solution for the discrete riccati equation. *De Gruyter Open*, 35(2):382–384, 2015.
- [6] Arduino Forum. Calculation speed tables. Online., 2013. [cit. 2019-04-17] <https://forum.arduino.cc/index.php?topic=200585.0>.
- [7] Arduino Playground. Matrix - library for arduino matrix calculation! ReadMe. Online., 2018. [cit. 2019-02-07]. <https://playground.arduino.cc/Code/Matrix>.
- [8] B. Earl. Memories of an arduino. User’s manual. Online., 2018. [cit. 2019-04-17]. <https://learn.adafruit.com/memories-of-an-arduino/arduino-memories>.
- [9] Bill Earl. Memories of an arduino. User’s manual. Online., 2018. [cit. 2019-03-30]. <https://cdn-learn.adafruit.com/downloads/pdf/memories-of-an-arduino.pdf>.
- [10] C. Belavý. *Teória Automatického Riadenia II: Návod na cvičenia*. Slovenská vysoká škola technická v Bratislave: Strojnícka Fakulta, Bratislava, 1990.
- [11] M. G. G. Takács. *Základy Prediktívneho Riadenia*. Slovenská technická univerzita v Bratislave, Bratislava, 2018.
- [12] GeeksforGeeks.org. Stack vs heap memory allocation. User’s manual. Online. [cit. 2019-04-17] <https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/>.
- [13] GitHub, Inc. Lela. ReadMe. Online., 2012. [cit. 2019-02-17]. <https://github.com/Singular/LELA>.

- [14] IEEE. IEEE standard 754 floating point numbers. IEEE Standard. Online. [cit. 2019-04-17] <https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/>.
- [15] QuantStart Team. Eigen library for matrix algebra in c++. Article. Online., 2017. [cit. 2019-02-07]. <https://www.quantstart.com/articles/Eigen-Library-for-Matrix-Algebra-in-C>.
- [16] R. A. Pavlik. Eigen for embedded use. ReadMe. Online., 2012. [cit. 2019-02-07]. <https://github.com/vancegroup/EigenArduino>.
- [17] Robert L. Williams II, Douglas A. Lawrence. *Linear State-Space Control Systems*. Willey and Sons, New Jersey, 2007.
- [18] T. Stewart . Basic linear algebra. ReadMe. Online., 2017. [cit. 2019-02-07]. <https://github.com/tomstewart89/BasicLinearAlgebra>.
- [19] D. R. Vaughan. A nonrecursive algebraic solution for the discrete riccati equation. March 1970.
- [20] A. J. L. W. F. Arnold. Generlized eigenproblem algorithms and software for algebraic Riccati equations. *IEEE Vol. 72, No 12*, 1984.
- [21] Y. Zheng. Implementation of discrete algebraic Riccati equation solver with lapack in C++ environment. *AAE 590 Report 2009 Fall*, 2009.